Apple IIGS
#105: We Interrupt This CPU...

Written by: Matt Deatherage May 1992

This Technical Note supplements the discussion of how interrupts generally work (or don't work) on the Apple IIGS found in the *Apple IIGS Firmware Reference*. It also discusses how to patch into the interrupt chain and when not to use software interrupts.

This Note is a Supplement

That's right, a supplement. This is not the definitive, end-all discussion of interrupts on the Apple IIGS. Most of the information you need to know is available, and has been for several years, in the *Apple IIGS Firmware Reference*. If you're going to write an interrupt routine, you **need** to read Chapter 6 of the *Firmware Reference*.

No excuses. If you don't have the book, buy it or borrow it. People who use your software don't want to hear a sad story about how you wanted to spend the money on a couple of CDs instead of preventing their machine from crashing.

If you haven't read Chapter 6 of the *Firmware Reference*, do so before continuing; the rest of this Note will make much more sense if you're familiar with the material covered in that chapter.

A Note About Timing

There are lots of times listed in this Note, concerning how fast certain kinds of interrupts must be serviced before they're lost. Please remember that all times listed are **ideal** times—actual times are likely to be shorter. For example, a maximum response time of a millisecond means you have one millisecond from the time the peripheral asserts the `/IRQ` line until the interrupt must be serviced. If interrupts are disabled for the first 750 microseconds (μs) of that, then your maximum response time is 250 μs. This is why we constantly remind programmers to keep interrupts disabled for absolutely the shortest time possible. Also, all times reflecting serial or AppleTalk interrupts already take into account the serial chip's internal 3-byte buffer.

So What the Heck Are All Those Vectors?

At first, looking at all those various vectors seems pretty darned intimidating. However, the structure becomes clearer when you think about interrupt priority.

Some microprocessors allow interrupt requests to have priorities—higher priority interrupts can interrupt lower priority ones. The 65816 doesn't have this capability, so the best the Apple IIGS can do is check possible interrupt sources in highest-priority-first order. For example, AppleTalk interrupts must always be processed extremely quickly—from the time an AppleTalk interrupt is asserted, someone must read the data from the SCC within a maximum of 104.167 μs or data can be lost. That's not very much time at all, especially considering that the system may have interrupts disabled, or may be running at 1 MHz speed when the interrupt fires.

Serial interrupts are next—at 19,200 baud, there's a maximum of 1.094 milliseconds to read data before it's lost. (Multiplication shows that 38,400 baud has a maximum of 547 μs, and 57,600 baud has a maximum delay of 273.5 μs. Not much at all.)

You'd hope the Interrupt Manager in ROM would be smart enough to service AppleTalk interrupts first and serial interrupts next, and in fact that's what it does. In fact, it services them so fast that not all the system information is saved before checking the hardware and dispatching (if necessary) to the `IRQ.APTALK` or `IRQ.SERIAL` vectors. See Apple IIGS Technical Note #24 for more information on which system state information isn't saved before calling those vectors.

The list of interrupt priorities is on page 180 of the *Firmware Reference*. What's not clear from any description of interrupt handling is that each internal interrupt source's vector is only called if the Interrupt Manager determines it is the source of the interrupt. For example, the `IRQ.DSKACC` vector is **not** called unless the user pressed Command-Control-Esc to generate the interrupt. This insures that external interrupt handlers for slot-based peripherals are dispatched to as quickly as possible—if each vectored routine had to determine interrupt ownership, every interrupt would have significantly more overhead.

There are two additions to the priority list in the *Firmware Reference*—the first is also an exception to the "interrupt handlers don't have to identify the interrupt" rule. On ROM 3 machines only, vector $E1021C (`IRQ.MIDI`) gets control immediately after determining the interrupt isn't an AppleTalk interrupt. MIDI data can come in so quickly that it needs higher priority than serial interrupts. However, to improve performance, routines called through this vector must return as fast as possible (faster would be better) to avoid delaying interrupts further down the chain, like serial interrupts. Also note that this vector doesn't exist on ROM 1.

The second addition is to the final priority, simply defined as "external slot." The documentation doesn't clearly indicate how this works—it kind of implies this is just calling `IRQ.OTHER`. In fact, if no `IRQ.OTHER` routine claims the interrupt, the system does some voodoo magic to switch to emulation mode and jumps through the vector at $03FE, just like all previous Apple II models. And just like in older systems, whatever code is pointed to by $03FE must end with an `RTI` instruction. This behavior is preserved for compatibility, although it is the slowest interrupt response available on the IIGS.

Getting Control In Time

Passing control to external handlers isn't always quick enough for some people. If you're writing a telecommunications program, for example, you have no more than 1.094 ms from the time a character is received to get it out of the SCC or you'll lose data at 19,200 baud.

The Interrupt Manager is a very tight piece of code—if it were running in RAM and the system was temporarily slowed down to 1 MHz, there would only be room for about two more instructions before AppleTalk would lose data. Since AppleTalk has to be serviced within 104.2 μs (as discussed previously), and since `IRQ.SERIAL` is called as quickly as possible after `IRQ.APTALK` (the only delay is if you're on ROM 3 and a non-trivial MIDI interrupt handler is installed), patching in at `IRQ.SERIAL` poses no problems for most high-speed communications, even up to 57,600 baud. In other words, it's not necessary to patch any vector other than `IRQ.SERIAL` to achieve the results you want.

The problem comes when you have external communications hardware—making it through the internal interrupt chain is too slow if your external communications hardware has the same kinds of limitation the SCC does (namely, a 3-byte internal buffer). External vectors are only called after all the internal sources verify it's not their interrupt, and by that time your card may have lost data.

Patching the Main Interrupt Vector

In these cases, where there is no possible way to service an interrupt in time through the Interrupt Manager's normal priority chain, and in these cases **only**, it's acceptable to patch out the main interrupt vector at $E10010 (preferably using `GetVector` and `SetVector` with reference number $0004). But even then, there are rules to follow.

1. You should duplicate the functionality of the main interrupt vector exactly until the point where you **must** gain control or lose data. For example, if your card requires that you service interrupts within a millisecond or lose data, AppleTalk interrupts still have higher priority over your interrupts because AppleTalk interrupts must be serviced within 104 μs. In this example case, your code should duplicate the functionality of the Interrupt Manager up through and including the call to `IRQ.APTALK`, and then (and **only** then) call your interrupt handler, where you handle the interrupt if it's yours and pass control to the rest of the interrupt chain if it's not.

2. You should only service your interrupts before AppleTalk if your interrupts require servicing in less than 104 μs. If they don't, give AppleTalk first shot. If they do, you must **clearly** inform the user, both in documentation and on the screen, that if they proceed with this function network services may be interrupted, and that they may have to restart the system to restore them. Users must also have the option to back out and cancel at this point. No, this isn't a pleasant message to deliver, but it's much nicer than to completely disconnect AppleTalk and lock up the system if it was booted from a server.

3. You should only patch out the main interrupt vector when absolutely necessary. For example, if you're communicating with hardware that runs at multiple speeds and only the highest speed generates interrupts that require patching the main vector, you should **not** be patching the main vector when not using that highest speed. For telecommunication programs, this means different interrupt handling routines depending on baud rates. To do this any other way lessens the reliability of other high-speed interrupt-driven peripherals in the system.

And remember, it's only acceptable to patch the main interrupt vector when there is no other way to service interrupts fast enough. At all other times, even in the same program, service your interrupts in other ways.

Vectors vs. Binding vs. Allocating

There are three main ways to get into the IIGS interrupt-handling chain—by patching vectors directly, by using the ProDOS 8 or ProDOS 16 call `ALLOC_INTERRUPT`, and by using the GS/OS call `BindInt`. Each behaves differently and has advantages and disadvantages. We'll go from the highest level to the lowest in discussing them.

BindInt—easy to use, but not as easy to control

BindInt's vector reference numbers (VRNs) are designed to correspond to vectors in the IIGS Interrupt Manager's chain. Comparing the list of numbers on page 265 of *GS/OS Reference* to the list of vectors starting on page 266 of the *Apple IIGS Firmware Reference* will make this more obvious.

When you call BindInt, GS/OS replaces the address in the appropriate interrupt vector with an address inside GS/OS. The routine it points to calls all the routines bound to that vector, including the one that was originally installed (usually the ROM's built-in SEC/RTL address). That is, if IRQ.VBL pointed to the Miscellaneous Tools' Heartbeat Task code before a program made four separate BindInt calls to VRN $000C, then after those calls completed, IRQ.VBL would point to code inside GS/OS that called all four bound routines and the Miscellaneous Tools' Heartbeat Task code.

This is why each bound routine is told (through the microprocessor's carry flag) if one of the other routines has already claimed the interrupt and why preserving that status is important. BindInt is a convenient way to get code time during various kinds of interrupts, but you should note that you can't control in what order bound handlers are called.

ALLOC_INTERRUPT—old style interrupt management

ALLOC_INTERRUPT and the ProDOS 8 equivalent, ALLOC_INT take the address of the routine you pass and keep it in an internal table. When an interrupt occurs, each address in the table is called in turn until one of the interrupt handlers claims it. In older days, failure by any of the installed interrupt handlers to claim the interrupt would bring the system to a crashing halt—nowadays unclaimed interrupts are ignored by both ProDOS 8 and GS/OS.

What the manuals **don't** tell you is that any routine installed in this way is called after the system has jumped through address $03FE in bank zero—in other words, at the last possible chance. For any kind of timing-sensitive interrupts, these routines are not sufficient.

The table that stores these routines is of a fixed size—ProDOS 8's table holds four routines, and GS/OS's holds 16. If you try to install more handlers than that, you'll get an error from the operating system.

Patching Vectors—high level of control, high risk

The lowest level at which you can get control is by directly patching the Interrupt Manager's vectors as documented in the *Firmware Reference*. Although this lets you get control as soon as the Interrupt Manager determines which vector to call, it also carries some compatibility risks.

Any BindInt calls with VRNs that reference a vector you patch make GS/OS take your routine's address and store it internally. This is a problem for anyone who daisy-chained into the same interrupt vector after you did—there's no good way to disconnect yourself without disconnecting everyone who patched in after you. This is Bad.

If you patch vectors directly, you have to check the vector when you're ready to remove your routine. If the vector doesn't still point to your address, someone else has patched into the vector after you and you can't remove yourself. In these cases, you have to leave a "code stub" that takes no action other than passing control along to the address that was installed when you patched in, and you have to leave that code stub at the same address as your interrupt handler. (Since you don't know who has patched the vector after you, you have no way to communicate with those programs and tell them you're going away.)

This means your interrupt handler can't be in your main program. If it is, when GS/OS calls `UserShutDown` to remove your program from memory, you'll orphan one or more pointers to your interrupt handler (which doesn't exist anymore). You must allocate memory and load your interrupt handler with a different user ID than your main program so your code stub can survive when your program quits. Also note that this means repeated launchings of your program could leave lots and lots of code stubs in memory—so if you can find a way other than patching vectors directly, you're encouraged to use it.

Software Interrupts—BRK and COP

Sometimes developers forget that `BRK` and `COP` instructions are in fact software interrupts—when the IIGS's 65816 encounters one of these instructions, it goes through the same Interrupt Manager procedures that all interrupts go through.

Among other things, this means that encountering one of these instructions inside an interrupt routine will overwrite all the system's saved information (such as registers or system state variables) with new ones, meaning you'll never be able to return from the first interrupt. This isn't too much of a problem with `BRK` (except when debugging interrupt routines), but a recent fad popularity for `COP` makes this worth mentioning.

Some developers are trying to use `COP` instructions for all kinds of general-purpose mechanisms, but the system is not designed to handle this. Using a `COP` instruction to pass control to a shell or a library routine in production-level code is not acceptable for several reasons. First, any `COP` instruction inside an interrupt handler will bring the system to its knees. Second, there is no arbitration for the `COP` vector so multiple users of it will collide. Third, although a `COP` instruction takes only two bytes, it takes many **hundreds** more cycles to execute than a `JSL` instruction, slowing the system down for no reason.

`COP` instructions are perfectly acceptable in non-production level (debugging) code, but developers should not use them as a way for different program modules to communicate. Such use is not supported and is strongly discouraged by Apple.

Before we RTI—A Summary

This Note covers many issues concerning interrupts, so here's a summary. This isn't all the explanation—refer to individual topics for discussions and reasons.

- Never disable interrupts for longer than necessary—you make life really difficult on routines that rely on high-speed interrupt capability.

- Interrupt routines should patch in as late as possible in the interrupt process without losing data. If your interrupt source doesn't need servicing as fast as AppleTalk does, don't patch in before AppleTalk.

- Patching the main interrupt vector at $E10010 is **only** acceptable if there's **no** possible way to service external interrupts quickly enough (internal interrupt sources, like serial ports, should always use other vectors), and even then the vector most only be patched while necessary; if a slower interrupt source is used in the same program, unpatch the vector.

- Different methods of installing interrupt handlers give you different levels of control. `BindInt` is the overall best method, although you can't control in what order bound routines are called.

- `COP` instructions are unacceptable in non-debugging code; they should never take the place of `JSL` instructions or other methods of inter-process communication.

Further Reference

- *Apple IIGS Firmware Reference*
- *Apple IIGS Toolbox Reference*, Volume 3
- *GS/OS Reference*
- *ProDOS 8 Technical Reference Manual*
- Apple IIGS Technical Note #24, *Apple IIGS Toolbox Reference* updates